8.1 Introduction

In this final chapter we shall tidy up a few loose ends, and discuss some strategies and issues with which a student of program testing should become familiar.

Programmers not only make mistakes in writing a program, they make mistakes in modifying an existing program as well. When a program is modified to remove an error or to add a new function, we need not only to test the program to make sure that the intended modification is properly done, we also have to ensure that no new error has been introduced into the program. In practice, this is done by performing the so-called regression testing described in Sec. 8.2.

To cope with complexity, programmers build software component by component. They then test each component in isolation to make sure that it will work as intended. That process is called unit testing. Most of the test-case selection methods discussed in Chapters 2 and 3 are intended for unit testing. What constitutes a unit in unit testing? Usually, it is the smallest separately compilable programming construct, such as a function in a C program. It may be of some other programming constructs, depending on the language and paradigm used.

Unit testing is to be followed by integration testing. Integration refers to the process in which program units are put together to form the system. There are certain types of error that cannot be revealed until the individually tested units are made to work together. Integration testing is designed to reveal such errors. It can be done incrementally by using different strategies as described in Sec. 8.3. Non-incremental approach, also known as "big bang" approach, is not advisable unless the program is very small because no one can expect a sizeable new program to be completely error free. When an error is discovered, big bang approach will provide very little clue as to the source of the error. It will be obvious if an incremental approach is taken: the source or error must be in the unit being integrated.

8.2 Regression Testing

Regression testing should be performed to a program after it is modified to remove an error or to add a function.

The process is to rerun all or some of the previous tests to assure that no errors have been introduced through the changes or to verify that the software still performs the same functions in the same manner as its older version.

Testing is done in the same sequence of stages as during development: unit testing, integration testing, system testing, acceptance testing.
8.3 Integration Testing

Integration testing is a process in which the components of a software system is integrated into a functioning whole. It can be done in two major ways.

Bottom-up integration:

This approach starts with unit testing, followed by subsystem testing, followed by testing of the entire system. Unit testing has the goal of discovering errors in the individual modules of the system. Modules are tested in isolation from one another in an artificial environment known as a test harness, which consists of the driver programs and data necessary to exercise the modules.

Advantages:

Unit testing is eased by a system structure that is composed of small, loosely coupled modules

Disadvantages:

1. The necessity to write and debug test harness for the modules and subsystems. Test harness preparation can amount to 50% or more of the coding and debugging effort.
2. The necessity to deal with the complexity that results from combining modules and subsystems into larger and larger units.

Top-down integration:

This approach starts with the main routine and one or two immediately subordinate routines in the system structure. After this top-level skeleton has been thoroughly tested, it becomes the test harness for its immediately subordinate routines. Top-down integration requires the use of program stubs to simulate the effect of lower-level routines that are called by those being tested.

Advantages:

1. System integration is distributed throughout the implementation phase; modules are integrated as they are developed.
2. Top-level interfaces are tested first and most often.
3. The top-level routines provide a natural test harness for lower-level routines.
4. Errors are localized to the new modules and interfaces that are being added.

Disadvantages:

1. Sometimes it may be difficult to find top-level input data that will exercise a lower-level module in a particular desired manner.
2. The evolving system may be very expensive to run as a test harness for new routines.
3. It may be costly to relink and re-execute a system each time a new routine is added.
4. It may not be possible to use program stubs to simulate modules below the current level.
**Sandwich integration:**

This approach is predominately top-down, but bottom-up techniques are used on some modules and subsystems. This mix alleviates many of the problems encountered in pure top-down testing and retain the advantages of top-down integration at the subsystem and system level.

**Phased Integration:**

In the phased approach, units at the next level are integrated all at once, and the system is tested before integrating all modules at the following level.

Unit testing is a point of contention often raised in debates over the strategies to be used in integration testing. Poston and Bruen [POBR87] state that unit tests "let the engineers locate the problem areas and causes within minutes." Beizer [BEIZ84] and Hetzel [HETZ88] also advocate unit test. On the other hand, Mills [MILL86], Yourdon [YOUR89], and Collins and Blay [COBL83] recommend integration strategies in which units are not tested in isolation, but are tested only after integration into a system.

Experiments conducted by Solheim and Rowland [SORO93] indicate that the top-down strategies generally produce the most reliable systems and are the most effective in terms of defect detection. They conjectured that the higher reliability is related to the fact that the higher level modules have a relatively high probability of being exercised by any given test case and are hence more important than lower level modules. The higher defect detection rate would appear to be caused by the fact that the top-down strategies exercise more modules per test case than do the other strategies.

If unit tests are about as costly as other types of tests, then the experiments suggest that one should not do spot unit test. On the other hand, if it is significantly cheaper to create and conduct unit tests than other types of tests, then the experiments suggest that spot unit testing is an effective way to correct software defects. Furthermore, the general question of whether or not to adopt unit testing might depend on factors not considered in the Solheim and Rowland's research. For example, Poston and Bruen felt that unit testing helps software engineers locate defects quickly, while Mills wrote that “unit debugging is a good way to inadvertently trade simple blunders for deep system errors through the tunnel vision of debugging.”

The experiments do not indicate a clear preference for either phased or incrementalal strategies. Again, factors that have not been considered in this research might weigh heavily when choosing between a phased and an incremental strategy. For example, if time is a great constraint on a project, then a phased strategy might be preferred since a phased strategy might proceed more quickly than an incremental strategy. On the other hand, incremental strategies might allow debuggers to locate defects more easily.

**8.4 Testing Object-Oriented Programs**
Most of the test methods discussed in the preceding chapters were developed based on the traditional function- or procedure-oriented paradigm. They have been found to be inadequate for testing object-oriented software.

To test an object-oriented program, we need to deal with problems introduced by the new features of an object-oriented programming language. Those features, such as encapsulation, inheritance, polymorphism, and dynamic binding, provide significant benefits in software design and programming. However, those new features also cause challenging problems in software testing and maintenance. Encapsulation means modeling and storing with an object the attributes and the operations an object is capable of performing. The interaction between two or more objects becomes implicit in the code. This makes it difficult to understand object interactions and properties inherited by its subclasses unless. Inheritance means properties defined for a class are inherited by its subclasses, unless it is explicitly excluded. However, a method that is tested to be correct in the context of the base class does not guarantee that it will work correctly in the context of the derived class. Polymorphism means the ability to assume more than one property, both in terms of data and operations. That is, an attribute of an object may refer to more than one type of data, and an operation may have more than one implementation. Dynamic binding means code that implements an operation is unknown until run time. These features make testing more difficult because the exact data type and implementation cannot be determined statically, and the control flow of an object-oriented program is less transparent.

**Impact of Encapsulation**

Encapsulation affects program testing in two ways:

- The basic testable unit will no longer be a subprogram, and
- Traditional strategies for integration testing are no longer applicable.

Here, by a subprogram, we mean a separately compilable program unit, such as a function, procedure, or subroutine. It is considered to be the basic building block of a program.

Once a subprogram is tested, it was seldom, if ever, tested as a unit again. If a subprogram unit was reused (either in the same application or in another application), its appropriateness had to be reassessed in the new context. For example, was the function being performed the right one for the given context, and did the interface for the module mesh smoothly with the surface?

For object-oriented environment, we are dealing with larger program units such as classes. Furthermore, the concept of subprogram is not quite the same as that in a traditional implementation. In particular, we tend to separate the specification (interface) for the subprogram from its implementation (body). We refer to the specification as a “method interface” (i.e., an advertised capability in the external interface the class presents to the outside world), and to the implementation as a “method” (i.e., the internal (hidden) algorithm by which the operation is carried out). We often complicate matters further by allowing one method interface to be supported by other methods.

Let us view a method interface plus one of its methods as the equivalent of a single subprogram in a traditional program. A class can therefore be thought of as encapsulating many subprograms. In
addition, the same subprogram may find itself being encapsulated in a number of different classes - frequently in descendants of the class in which it appeared first. We also stipulate that classes may also encapsulate exceptions, other objects, and various forms of state information.

In object oriented systems, we will most often think of subprograms as being bound (encapsulated) within a larger entity, e.g., a class. Furthermore, these subprograms will work in conjunction with other items encapsulated within the same object. This means that, in an object-oriented environment, attempting to test a subprogram in isolation is virtually meaningless. In fact the smallest testable unit is no longer a subprogram, but a class or an instance of class.

Weyuker has proposed 11 axioms for checking the completeness of a test [FRWE88]. Weyuker’s fifth axiom is about antiextensionality. It says that, for two different algorithms that compute the same function (i.e., they are “semantically identical”), an adequate test set for one algorithm is not necessarily an adequate test set for the other. This says that, if we replace an inherited method with a locally defined method that performs the same function, the test set for the inherited method is not guaranteed to be adequate for the locally defined method.

Weyuker’s sixth axiom is referred to as the “general multiple-change” axiom. This axiom says that if two programs are of the same shape (i.e., if one can be transformed into the other through a simple replacement of one or more relational operators, one or more constants with other constants, and one or more arithmetic operators with other arithmetic operators.), a test for one will not necessarily be adequate for the other. This axiom tells us that when the same items are inherited along different ancestor paths (i.e., via changes in the precedence ordering in a multiple inheritance scheme) then different test sets will be needed.

Weyuker’s seventh axiom is called the antidecomposition axiom. This says that something tested in one context will have to be retested if the context changes. For example, suppose that I thoroughly test a method within the context of a given class. Next suppose I create a specialization (e.g., a subclass or a derived class) based on this class and that the specialization inherits the tested method from the generalization (e.g., a superclass or base class). Even though I have tested the method within the context of the generalization, I cannot begin to guarantee the appropriateness of the same method within the context of the specialization, unless I re-test the method within the context of the specialization.

Weyuker’s eighth axiom is the anticomposition axiom. It says that adequately testing each unit in isolation is usually insufficient to be considered as the same as adequately testing the entire (integrated) program. Suppose that we change the underlying implementation for a given object, but keep the interface constant. We might suspect that we could get away with simply retesting the modified object in isolation. This is not the case. We will have to retest all dependent units (e.g., specializations and units that directly reference the modified object) as well.

Integration Testing

There are two major ways to do integration testing: incremental and non-incremental. For a traditional program, once a unit (usually a subprogram) is tested in isolation, it is then integrated into a larger subsystem. Non-incremental integration means that each unit is tested in isolation, and then simultaneously integrate all units and then test the resulting whole. (This is also known as
the "Big Bang" approach to integration testing.) This approach may work well for small systems, but is not advisable.

Incremental testing, on the other hand, dictates that each unit be tested in isolation, and then integrate each unit, one at a time, into the system. There are three different approaches to incremental testing, viz., top-down, bottom-up, and sandwich. Incremental testing is preferable to non-incremental testing because it would be easier to localize the problem once an error is detected.

For an object-oriented approach, the smallest testable unit will be a class (or a comparable object-oriented program unit). Given that the methods associated with each method interface often take advantage of the underlying implementation, it is difficult to see how each method can be tested in isolation.

Integrating “subprograms” into a class, one at a time, testing the whole as we go, may not be an option. For example, there are usually direct or indirect interactions among the components that make up the class. One method may require that the object be in a specific state, which can only be set by another encapsulated method, or a combination of encapsulated methods. Reversing the order of integration may not help since each method may be used to test the other.

There are other implications to integration testing in an object-oriented environment. For example, in a traditional software system, existence of invocation hierarchies (module A calls module B, which eventually returns control to module A etc.) is the norm. An object-oriented system is not required to abide by a strict invocation hierarchy. For example, object A may interact with object B without necessarily having to go through an intermediary (controlling) object. This has led some people to say things such as “object-oriented systems have no tops”, i.e., there will not be a master control module at the apex of some invocation hierarchy.

If we are working with a system that “has no top”, then it will be difficult to define such things as “top-down integration testing”, “bottom-up integration testing” etc. All of these testing strategies assume that there must be a definite “top” and “bottom” for the system. This implies that new integration testing method have to be developed.

The Impact of Information Hiding on Testing

Advocates of object-oriented technology are fond of citing the black-box nature of objects because a user of an object is denied access to the underlying implementation. This creates problems during testing. For example, consider a simple list object. In its interface there are a number of methods, e.g., add, delete, length, etc. Suppose we add an item to the list using the add method. How can we be sure that the specific item was actually added? Since we cannot inspect the underlying implementation, we must seek some other strategy.

A general approach is to first establish an acceptable level of confidence in the state-reporting methods in the object’s interface, i.e., those methods that do not change or alter the state of the object, but rather return information on some aspect of the object’s state. For example if we could trust the length method in our list object, we could use it to test the "add" and "delete" methods by looking at the length of the list after invocation of these methods.
Much of what we know about testing technology does indeed apply to object-oriented systems. Object-orientation, however, brings with it its own set of concerns.

**Unit Testing**

Software unit testing is performed to find errors in a software component, such as a module. Unlike a traditional program where functional modules and functional procedures are considered as its components, an object-oriented program consists of three levels of components: (1) functions defined in classes, (2) classes, and (3) modules consisting of a collection of classes.

Although there are many research results addressing unit-testing problems, most of them focused on procedure-oriented software. The test-case selection methods for unit testing can be classified into two types: program based and specification based. The program based methods establish test-case selection criteria based on the source-code of the program whereas the specification based methods establish test-case selection methods based on the program specifications.

Since classes are the major components in an object-oriented program, testers have to find answers to the following questions:

- Can existing unit testing techniques be applied to objects and classes?
- What test criteria can be used in unit testing for object-oriented software?
- How can unit test for an OO program be performed in a systematic way?

Parrish et al. [PABC93] developed a fundamental theory for application of conventional flow-graph-based test methods to OO class modules. Based on the conventional flow-graph model, they proposed a general class-graph model. Many existing program-based and specification-based test-case selection methods become applicable to a class when it is represented by a class graph. The authors provided their insights on how to use this class graph to define a new set of test coverage criteria, and how to select test cases to satisfy them. This makes the results more applicable to class testing in practice as explained elsewhere.

### 8.5 Flow Graph-Based Testing of Object-Oriented Software Modules

There are several techniques for testing classes individually, i.e., for unit testing. Most of them involve in sending a sequence of messages constructed based on the specification to an object of the class and observe the resulting effect on the object.

To fix the idea, consider the C++ code listed in Figure 8.5.1, 8.5.2 and 8.5.3 giving the definition (header file), implementation (member functions) and a driver file (utility function), respectively, for a Stack Class.
// Definition of a Stack Class
// STACK.H

#ifndef STACK_H
#define STACK_H
#include <iostream.h>
class Stack {
public:
    Stack (int = 10);    // default constructor (stack size 10)
    ~Stack() { delete stackPtr; } // destructor
    int push (const T&);   // push an element onto the stack

private:
    int size;   // # of elements in the stack
    int top;   // location of the top element
    T *stackPtr;  // pointer to the stack
};

Figure 8.5.1: Stack Class Header file

// Stack Class Member Functions
// STACK.CPP

#include <iostream.h>
#include "Stack.h"

// Constructor with default size 10
Stack :: Stack (int s)
{
    size = s > 0 && s < 1000 ? s : 10; //size is 10 if not between 0 and 1000
    top = -1;     // stack is initially empty
    stackPtr = new T[size];  // space allocated for stack elements
}

// Push an element onto the stack
int Stack :: push (const T &item)
{
    if (!isFull()) {
        stackPtr[++top] =item;   // place item in Stack
        return 1;    // push successful
    }
    return 0;     // otherwise push is unsuccessful
}

.
.
.

Figure 8.5.2: Implementation of a Stack Class
// Utility function for a Stack Class
// DRIVER.CPP

#include <iostream.h>
#include "Stack.h"

main()
{
    Stack myStack (5);
    int i;
    cout << "Pushing elements onto the stack" < endl;

    while ( myStack.push(i) ) {
        cout << i << ' ';
    }
    cout << "Stack is full" << endl;
    return 0;
}

Figure 8.5.3: Utility Function for a Stack Class

In the above figures, we have created an object named myStack. This will instantiate a stack named myStack, and will invoke the constructor of the Stack Class. It will pass the (message) number 5 as the size of the stack. Every call of the push function will pass an integer as a message to the push function.

The test-case generation techniques for classes fall into two categories: random or systematic. A random technique generates message sequences entirely at random; most existing techniques use this approach. A systematic technique uses a repeatable procedure to generate a fixed sequence satisfying certain properties. Systematic techniques have the advantage of guaranteeing that specific characteristics of a class have been examined in a standard manner, regardless of who applies the technique. For example, a trivial systematic technique would require that every message associated with a particular class be applied at least once to some object.

Systematic testing can be applied to conventional software as described in Chapter 2. Those techniques associate a (control or data) flow graph with every module and require test data to achieve a certain level of coverage over the flow graph. Zweben et al. [ZWHK92] showed that it is possible to associate a flow graph with a class module. A node in such a flow graph represents a message; and an edge, say, from node A to node B, represents the possibility that A might be sent, followed by B. Using this notion of a “class-flow graph”, a collection of systematic coverage criteria for class modules can be defined.

The use of flow graph to model class behavior provides a wide range of class-testing techniques that have already been analyzed and exploited for conventional programs. The analysis of flow graph-based testing techniques for conventional programs provides a model for the types of analyses that might be useful for class testing. In addition, the experience that many testing practitioners have acquired through the use of flow graph-based techniques for conventional software will likely facilitate effective technology transfer with respect to the analogous class-testing techniques.
In the following we will present a general conceptual framework for conducting flow graph testing of classes. In this framework, we clarify a number of concepts related to how this modeling should be performed, particularly with regard to the feasibility or traversability of an edge. The idea of feasibility is universally well understood in the context of flow graph modeling of conventional programs, but there are complicating issues in the context of classes. These techniques may be applied even in the absence of formal specification (model based), thus making them applicable to current-day development practice.

Based on the expanded conceptual framework, we propose new class-based analogs of conventional specification-based techniques that can be automated, in the sense that the techniques may be used to automatically generate sequences of test messages. According to the authors, these techniques do not necessarily ensure a test suite that is as rigorous as those produced by manual techniques. This will provide an intermediate result, in the sense that the tester can use these techniques to reach a basic level of test coverage satisfaction, and then apply manual techniques to augment the test suite in order to achieve a better coverage.

Here the term "class" is used to refer to the implementation of an abstract data type within an object-oriented language. As we have seen, an object is an instance of a class. A class implementation is defined in two parts: an interface (definition) and a body (implementation). Operations are sometimes called methods; invoking a method with respect to a given object is sometimes referred to as “sending a message” to the object.

Two formal specification techniques – algebraic specification and model-based specifications – are commonly discussed in the literature. Algebraic specification defines the behavior of a module in terms of a set of axioms that characterize the equivalence of combinations of operations, whereas model-based specifications involve the individual modeling (using well-defined concepts) of each operation in the class. Table 8.5.1 presents both algebraic and model-based specifications for a stack class.

In Table 8.5.1 the algebraic specifications are composed of three elements: the function section defines the signatures of the class operations, the domain conditions prescribe the restrictions to be placed on the input values for a particular operation, and the axioms are the specifications that must be satisfied by the implementations.

For model-based specification, each operation identifies both a "requires" and an "ensures" clauses. The "requires" clause asserts any domain restrictions on the operations, similar to the domain condition section of the algebraic specification. The "ensures" clause indicates a specific condition that holds true after invoking the operation, provided that the "requires" clause is satisfied. A well-defined concept from mathematics, the string theory is used to model a stack. Here $e$ denotes an empty string, $o$ the operation of concatenation, and $\#x$ the old value of $x$. 
Table 8.5.1. Formal Module Specification

<table>
<thead>
<tr>
<th>Algebraic</th>
<th>Model-Based</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>module STACK_TEMPLATE(type T)</strong></td>
<td><strong>module STACK_TEMPLATE (type T)</strong></td>
</tr>
<tr>
<td><strong>type STACK</strong></td>
<td><strong>type STACK is modeled by STRING</strong></td>
</tr>
<tr>
<td><strong>functions</strong></td>
<td><strong>interface</strong></td>
</tr>
<tr>
<td>NEW: → STACK</td>
<td>operations New (s: STACK)</td>
</tr>
<tr>
<td>PUSH: STACK × T → STACK</td>
<td>ensures s = e</td>
</tr>
<tr>
<td>POP: STACK → STACK</td>
<td>operations Push (s: STACK, x:T)</td>
</tr>
<tr>
<td>TOP: STACK → T</td>
<td>ensures (s = #x ° #s) ∧ (x = #x)</td>
</tr>
<tr>
<td><strong>domain conditions</strong></td>
<td><strong>operations POP (s: STACK)</strong></td>
</tr>
<tr>
<td>POP(s): not (s = NEW)</td>
<td>requires not (s = e)</td>
</tr>
<tr>
<td>TOP(s): not (s = NEW)</td>
<td>ensures ∃x ∃ #x = x ° s</td>
</tr>
<tr>
<td><strong>axioms</strong></td>
<td><strong>operations Top (s: STACK, x:T)</strong></td>
</tr>
<tr>
<td>(1) not (PUSH(s, x) = NEW)</td>
<td>requires not (s = e)</td>
</tr>
<tr>
<td>(2) POP (PUSH(s, x)) = s</td>
<td>ensures ∃t ∃ s = x ° t) ∧ (s = #s)</td>
</tr>
<tr>
<td>(3) TOP (PUSH(s, x)) = x</td>
<td><strong>end STACK_TEMPLATE</strong></td>
</tr>
</tbody>
</table>

As mentioned previously, our discussion of flow graph-based testing follows the work of Zweben et al. [ZWHK92], whose flow graph model can be stated as follows. A node in the graph represents an operation; an edge between operations A and B means that it is permissible for a client module to invoke operation A followed by operation B. Determination of whether or not an edge exist is based on the model-based specification for the class, i.e., on whether the "requires" clause of B can be satisfied by the "ensures" clause of A. In the previous example, there would be no edge from NEW to POP because NEW returns an empty stack, thus violating the "requires" clause of POP.

Certain operations are control operations, i.e., they return true or false and can be used to construct conditions in client modules. For such operations, there are two labeled edges emanating from them to every operation that can follow them in a client program. For example, for the control operation IS EMPTY found in some stack classes, there is a "true" and a "false" edge emanating from it to an operation that a client is permitted to perform (based on the ensures-requires conjunctions in the specification). Control operations thus have the same role in the class flow graph as conditions have in a conventional program flow graph.

A test-case selection criterion formulated for conventional program flow graph can now be replaced by one that defines sequences of operations based on the class flow graph. The class analog for node coverage requires that every operation be invoked, and the class analog for branch coverage requires that a sequence (or a set of sequences) containing every edge in the class flow graph be traversed during the test.

Class analog for the data-flow criteria can be provided similarly provided that the underlying data flow concepts (i.e., definitions, uses, def-use associations, etc.) are defined for classes as follows. First, all definitions and uses are associated with class operations rather than statement blocks. A p-use is a use associated with a control operation rather than a condition. Because there are
multiple labeled edges emanating from control operations, all of the concepts regarding p-uses are the same as before. In particular, each labeled edge emanating from a control operation represents a p-use to be exercised. A c-use is a use within any other operations.

The other general observation that we make about definitions and uses in this context is that data flow concepts must be interpreted in the context of type rather than that of variable. Operations define and use their parameters, e.g., a PUSH operation defines and uses any actual stack instance with which it is parameterized in a client module. Thus, in this example, data flow analysis takes place with respect to type ‘stack’ in general, rather than with respect to a specific variable.

In particular, definitions and uses can be interpreted as described in [ZWHK92] as follows. An operation is said to contain a definition of a type if a formal parameter to the operation appears in that type and the "ensures" clause specifies that the parameter is to be changed. For instance, if the symbol x appears without # in the "ensures" clause and the predicate x = #x is absent, then it is assumed that x is defined by the operation (the absence of # means that there is a new value for x). An operation is said to contain a use of a type if a formal parameter to the operation appears in that type and the "ensures" clause specifies that the parameter is used somehow to produce a result. For instance, if the symbol #x appears in the "ensures" clause in any predicate other than x = #x (which is considered neither a definition nor a use of x), then x is considered to have been used (since its old value is a part of some predicate that represents the computation of the operation).

These concepts can be clarified by the particulars listed below:

<table>
<thead>
<tr>
<th>The operation of</th>
<th>defines</th>
<th>uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEW</td>
<td>stack</td>
<td>&lt;&gt;</td>
</tr>
<tr>
<td>PUSH</td>
<td>stack, T</td>
<td>stack</td>
</tr>
<tr>
<td>POP</td>
<td>stack</td>
<td>stack</td>
</tr>
<tr>
<td>TOP</td>
<td>T</td>
<td>stack</td>
</tr>
</tbody>
</table>

Let us consider the model-based specification of a stack module presented earlier. This module has four operations: NEW, PUSH, POP, and TOP. Because of domain restrictions on POP and TOP, there is no edge leading from NEW to either of these operations. Otherwise, every pair of operation is connected by a single edge and there is a reflexive edge from every operation to itself as shown in figure 8.5.4.

Thus the sequence NEW, PUSH, TOP, POP will cause every node to be exercised, the sequence NEW, NEW, PUSH, NEW, PUSH, PUSH, POP, PUSH, TOP, TOP, PUSH, POP, POP, POP, NEW, PUSH, PUSH, PUSH, POP, TOP, POP, TOP, POP, TOP, NEW will cause every edge to be traversed, the sequence NEW, PUSH, PUSH, POP, TOP, PUSH will cause every define operation to be exercised during the test, and so on, and so forth.
Figure 8.5.4: Flow graph for a simple stack module.